AUTOGENICS: Automated Generation of Context-Aware Inline Comments for Code Snippets on Programming Q&A Sites Using LLM

Suborno Deb Bappon, Saikat Mondal, Banani Roy
Department of Computer Science, University of Saskatchewan, Canada
{suborno.deb, saikat.mondal, banani.roy}@usask.ca

***Abstract*—Inline comments in the source code facilitate easy comprehension, reusability, and enhanced readability. However, code snippets in answers on Q&A sites like Stack Overflow (SO) often lack comments because answerers volunteer their time and often skip comments or explanations due to time constraints. Existing studies show that these online code examples are difficult to read and understand, making it difficult for developers (especially novices) to use them correctly and leading to misuse. Given these challenges, we introduced AUTOGENICS, a tool designed to integrate with SO to generate effective inline comments for code snippets in SO answers exploiting large language models (LLMs). Our contributions are threefold. First, we randomly select 400 answer code snippets (200 Python + 200 Java) from SO and generate inline comments for them using LLMs (e.g., Gemini). We then manually evaluate these comments' effectiveness using four key metrics: accuracy, adequacy, conciseness, and usefulness. Overall, LLMs demonstrate promising effectiveness in generating inline comments for SO answer code snippets. Second, we surveyed 14 active SO users to perceive the effectiveness of these inline comments. The survey results are consistent with our previous manual evaluation. However, according to our evaluation, LLMs-generated comments are less effective for shorter code snippets and sometimes produce noisy comments. Third, to address the gaps, we introduced AUTOGENICS that extracts additional context from question texts and generates context-aware inline comments. It also optimizes comments by removing noise (e.g., comments in import statements and variable declarations). We evaluate the effectiveness of AUTOGENICS-generated comments using the same four metrics that outperform those of standard LLMs. AUTOGENICS might (a) enhance code comprehension with context-aware inline comments, (b) save time, and improve developers' ability to learn and reuse code more accurately.**

***Index Terms*—Stack overflow, Inline comments, Large Language Models, Tool Support, User Study**

Fig. 1: An overview of the AUTOGENICS workflow.

I. Introduction

Source code comments are essential for enhancing code comprehension, facilitating software maintenance, and promoting reusability [1, 2, 3, 4, 5, 6, 7, 8]. One key issue with code snippets found in answers on programming Q&A sites, such as Stack Overflow (SO), is the lack of comments. Two factors that might contribute to skip commenting on code snippets are the voluntary nature of SO participation and developers' time constraints [9, 10, 11]. Additionally, the manual process of adding comments to code is tedious, discouraging developers from doing so [12, 13]. Previous research shows that SO code examples often need better usability (e.g., readability) [14]. Answerers often do not provide adequate explanations for their code [14]. However, undocumented code is a major source of developer frustration, as developers often get confused while reading code snippets without proper comments [15, 16]. More than $582K$ answer code snippets have neither inline comments nor explanations [17]. Such evidence raises severe concerns about correctly reusing the code examples, potentially leading to misuse. Therefore, there is a growing demand for automated tools to generate comments for code snippets [18, 19, 20, 21, 22, 23, 24].

A few studies focus on method-level documentation to summarize the purpose and functionalities of methods [25, 26, 27, 28]. However, method-level comment generation techniques might not be suitable for SO code snippets, as these snippets often consist of a few bare statements rather than complete

methods [29]. Therefore, inline comments are more suitable in our target context. They clarify the functionalities of each line of code, making it easier to read and understand. Given these points, an investigation is warranted to find a better way to generate effective inline comments for SO answer code snippets that are frequently shared and reused. As far as we know, such investigation has yet to be addressed in existing literature.

Recent advancements in AI, especially LLMs, have revolutionized NLP tasks and achieved state-of-the-art results. LLMs with proper in-context learning and adequate prompts demonstrate superior performance in diverse software development tasks, including documentation [30, 31, 32, 33]. In this study, we thus leverage the power of LLMs to generate inline comments on the code snippets included with SO answers. We randomly select 400 code snippets (200 Java + 200 Python) found in SO answers. First, we generate inline comments for the code snippets using `Gemini 1.5 Pro`, which is free. We focus solely on standalone code snippets, excluding context (e.g., question descriptions), to evaluate how effectively the language model can generate comments. Then, we manually evaluate four key metrics - accuracy, adequacy, conciseness, and usefulness - to ensure the effectiveness of the generated inline comments. We randomly selected 20 samples and then generated inline comments using `GPT-4` [34] to see how consistent the comments were across different LLMs. We further conducted a user study to hear from the practitioners about the effectiveness of the LLM-generated comments. Fourteen professional software developers (who are also active users of SO) participated in this survey. As a practical outcome of our research, we introduced `AUTOGENICS`, a tool specifically designed to integrate with SO. Figure 1 presents the workflow of `AUTOGENICS`, which involves three straightforward steps: (1) activating the `AUTOGENICS` tool as a browser plugin, (2) locating the "AUTOGENICS" button below the code snippets in SO answers, and (3) generating inline code comments by clicking "AUTOGENICS". `AUTOGENICS` can generate inline comments by considering the context of the questions. It also optimizes comments by removing noise, making it a valuable assistance for software developers.

In this study, we answer three research questions and thus make three major contributions.

RQ1. How effective are LLMs at generating inline comments for code snippets found in Stack Overflow answers? This research question aims to evaluate the capability of LLMs (e.g., `Gemini`) to improve code comprehension by generating effective inline comments for SO answer code snippets. We evaluate the effectiveness of the LLM-generated inline comments by measuring their accuracy, adequacy, conciseness, and usefulness. We employ a 5-point Likert scale to quantify these metrics. Overall, LLMs demonstrate promising effectiveness. The accuracy, adequacy, and usefulness of LLM-generated inline comments improve with longer code snippets. The performance of the `Gemini 1.5 Pro` model closely matches that of `GPT-4`.

RQ2. How do developers perceive the effectiveness of these inline comments? Are they interested in an automated tool to generate them? Understanding how developers perceive the effectiveness of inline comments is crucial for enhancing the usability and impact of automated tools. We randomly selected eight example code snippets (four Java + four Python) with inline comments and asked the participants to evaluate their effectiveness using the same four metrics as in RQ1. Our user study confirms the results from RQ1. Approximately 79% of participants showed strong interest in an automated tool for generating inline comments.

RQ3. Can we introduce a tool into Stack Overflow to generate context-aware, noise-free, inline comments for code snippets? Are these comments more effective than those from standard LLMs? We explore the feasibility of introducing tool support to enhance code readability and understanding by automating the generation of context-aware inline comments for code snippets in SO answers. We then introduce `AUTOGENICS` to assist developers. `AUTOGENICS` can be easily integrated with the SO site to annotate code snippets with context-aware, noise-free inline comments, outperforming those generated by standard LLMs.

Replication Package available in our online appendix [35].

Answer to Question: **Spark handle json with dynamically named subschema**

```
with open("file_path.json", "r") as f:
    json_string = f.read()
    json_as_dict = json.loads(json_string)
    list_of_dicts = list(json_as_dict.values())
    df = spark.createDataFrame(list_of_dicts)
```

Comments

Why don't you explain code a bit more? – Commented on May 23, 2023 at 22:13

Your answer could be improved with additional supporting information. Please edit to add further details, such as citations or documentation, so that others can confirm that your answer is correct. – Commented on May 23, 2023 at 22:13

(a) An answer where users requested an explanation of the code to determine its correctness.

Answer after generating comments using **AUTOGENICS**

```
with open("file_path.json", "r") as f:
    # Read the entire JSON string
    json_string = f.read()
    # Convert the JSON string to a Python dictionary
    json_as_dict = json.loads(json_string)
    # Extract the values (user objects) from the dictionary as a list
    list_of_dicts = list(json_as_dict.values())
    # Create a Spark DataFrame from the list of user objects
    df = spark.createDataFrame(list_of_dicts)
```

(b) Answer after adding comments using `AUTOGENICS`.

Fig. 2: A motivational example [36] where users requested a code explanation to validate its accuracy contrasted with the same answer improved by `AUTOGENICS`-generated comments.

II. Motivational Example

Code segments submitted as part of answers on SO often lack inline comments and are not always adequately explained [14]. Let us consider the example answer in Fig. 2a. It contains only a Python code snippet. However, the answerer provided neither an explanation nor any comments. As a result, users who were trying to use it to resolve their problems struggled to understand it. A lack of understandability can lead to the

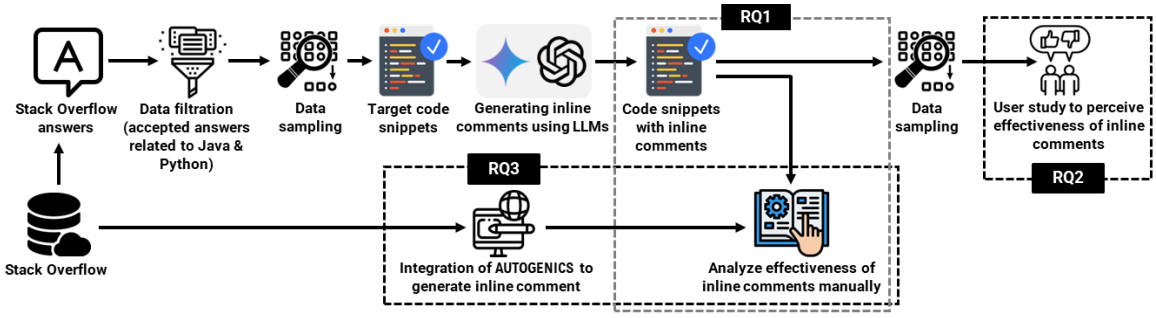


Fig. 3: Research methodology for human-centric evaluation of inline code comment generation.

TABLE I: Summary of our dataset (**Q1-Q4**: *Quartiles 1-4*).

	Total Answers	Accepted Answers with Single Code Fragment	Q1 (1<= LOC<= 2)	Q2 (3<= LOC<= 7)	Q3 (Python: 8<= LOC<= 14, Java: 8<= LOC<= 16)	Q4 (Python: 15<= LOC<= 695, Java: 17<= LOC<= 997)	Sampled Answers
Python	1606,298	470,393	121,065	135,144	101,011	112,302	200
Java	1357,200	393,684	98,883	108,164	93,513	92,258	200
Total	2963,498	864,077	219,948	243,308	194,524	204,560	400

misuse of code snippets and the introduction of latent bugs in production code. One user thus commented, "Why don't you explain code a bit more?". Another user requested documentation to determine if the answer was correct. Unfortunately, the answer score is zero, which means it is non-useful.

On the other hand, consider the same code snippet with inline comments generated by our tool `AUTOGENICS`, as shown in Fig. 2b. Each line of code is clearly explained, revealing its function and purpose. These comments make the code easy to understand, even for novice developers. Such comments significantly (a) reduce the time and effort needed to reuse the solution correctly and (b) enhance developers programming knowledge.

This example is one of many that motivates our study. In this study, we attempt to generate inline code comments by introducing tool support. Our tool can (a) assist millions of SO users in generating comments for code snippets in SO answers and (b) enhance the readability and understandability of code.

III. Study Methodology

Fig. 3 shows the schematic diagram of our study. We first collect about three million answers from SO. We collect these answers from two popular programming languages employing one restriction. We then randomly select 400 answers (200 Java + 200 Python) and generate inline comments utilizing LLMs. Then, we manually evaluate their quality from different aspects using four popular metrics. Second, we study 14 practitioners, as their opinions are crucial in determining the effectiveness of the inline comments. Finally, we introduce a tool that can be integrated with the SO site to generate context-aware inline comments and assess their quality manually. The following sections discuss different steps of our methodology.

A. Dataset Preparation

Data Collection. Table I lists the dataset for our study. We collected a total of 2963,498 answers from SO using StackExchange Data API [17], all of which were posted on or before February 2024. In particular, we collected these answers to questions related to two widely used programming languages, Python and Java, to generate inline comments using LLMs. This choice also allows us to assess LLMs' capability to generate comments for various programming language types, including static and dynamic ones. We choose answers under a restriction: the answer must be accepted by the question's owner and contain one code snippet. Accepted answers are widely regarded as reliable solutions. Focusing on one code snippet ensures consistency and avoids potential complications from merging multiple examples. After that, we get 470,393 Python answers and 393,684 Java answers. Finally, we extract code snippets using specialized HTML tags such as `<code>` under `<pre>`.

Quartile Analysis. We calculate the number of lines (i.e., LOC) of the code snippets and divide them into four quartiles. A quartile divides a dataset into four equal parts, each with 25% of the data samples. This approach allows us to evaluate the effectiveness of LLMs across code snippets with varying LOC. Table I shows the quartile-wise distribution of our code snippets. Interestingly, the number of code snippets across four quartiles for Python and Java are evenly distributed.

Data Sampling. Our dataset is meticulously prepared, ensuring fairness and representativeness. We randomly select 50 code snippets from each quartile, resulting in a final dataset of 400 code snippets, 200 from each Python and Java. Note that this sample size is statistically significant with a 95% confidence level and a 5% error margin [37, 38].

B. Inline Comments Generation Using LLMs

We primarily utilize Google's `Gemini 1.5. Pro` [39] to generate inline comments for our selected code snippets. We opted for the `Gemini` model for two key reasons - (1) it is freely accessible (offers 50 requests per day), thereby enabling easy replication of our findings, and (2)it demonstrates high

performance in several software development-related tasks (e.g., code generation) compared to other state-of-the-art models like GPT-4 [40]. According to our analysis, most code snippets in accepted answers lack classes or methods. However, a few of them include these structural elements. LLM models often generate both inline comments and method/class-level documentation for these code fragments. Surprisingly, sometimes, they modify the original snippets. Additionally, some code snippets may contain inline comments. Careful prompt design is crucial for retaining the original code snippets and adding inline comments (when necessary). Therefore, we consider these factors when designing effective prompts to ensure the precise generation of inline comments. Consider the following example prompt that was passed to Gemini to generate inline comments. The prompt includes three clear instructions with the target code snippet - (1) the purpose of inline code comments, (2) the direction not to alter the given code snippets, and (3) guidance to avoid class/method-level documentation (if there is any).

Inline comment generation prompt (Gemini): Given the following code snippet: {CODE_SNIPPET}. Generate inline comments to explain what each part of the code does. An inline comment is a single-line comment typically used to explain or clarify a specific line of code. It starts with // for Java and # for Python. Ensure that you only generate inline comments and do not alter the existing code. Avoid adding any comments at the class or method level; focus only on inline comments.

In addition to Gemini, we conducted a case study with GPT-4 to generate inline comments for a subset of 40 randomly selected code snippets (20 Python + 20 Java evenly distributed across quartiles). We used the same prompt as Gemini to ensure a fair evaluation. This approach allows us to explore GPT-4's capability to produce inline comments. Please note that we do not consider question contexts in this phase while generating inline comments.

C. Effectiveness Evaluation of Inline Comments

TABLE II: Inline comment evaluation metrics.

Metric	Description
Accuracy	The extent to which the generated inline comments correctly describe the functionality and behavior of the code.
Adequacy	The degree to which the inline comments provide sufficient information to understand the code without unnecessary details.
Conciseness	The measure of how brief and to-the-point the inline comments are, avoiding unnecessary verbosity.
Usefulness	The overall helpfulness of the inline comments in enhancing the comprehension and maintenance of the code.

Two authors, one with more than seven years and another with over 14 years of professional software development experience in Python and Java, manually evaluate the effectiveness of the inline comments generated by the LLMs. Previous studies have emphasized the importance of human evaluation for code comments, arguing that automatic metrics like BLEU and ROUGE may not always be reliable [1, 41, 42, 43, 44, 45]. Additionally, the lack of ground truth for inline comments on SO code snippets restricts us from utilizing such automated metrics.

We evaluate effectiveness using four metrics: accuracy, adequacy, conciseness, and usefulness (see descriptions Table II) [1]. We use a 5-point Likert-scale [46, 47] to quantify the assessment of each of the metrics. A higher rating signifies better quality of inline comments and vice versa. Initially, we conducted multiple interactive sessions to discuss and reach a consensus on the evaluation metrics. We then randomly select 80 code snippets with inline comments (40 Python + 40 Java, ten from each quartile) from our selected 400 code snippets. Next, we meticulously evaluate inline comments and assign rankings (1-5) based on the four metrics. We categorize ranks 1-3 as low and 4-5 as high. We then measure the agreement using Cohen's Kappa [48, 49]. The value of κ was 0.94, which means the strength of the agreement is almost perfect. Next, we resolve the remaining few disagreements by discussion. However, the agreement level indicates that any coder can do the rest of the ranking without introducing individual bias. Thus, the first author of this paper evaluates and ranks the remaining samples. We spent a total of 100 person-hours for this manual evaluation.

D. Developers' perception of the effectiveness of inline comments

Survey Design. We survey software practitioners to hear how they perceive the effectiveness of the LLM-generated inline comments. We follow the guidelines and steps outlined by Kitchenham and Pfleeger [50] for conducting personal opinion surveys. We also consider ethical issues from the established best practices [51, 52]. For instance, we obtained participants' consent, ensured the confidentiality of their information, and explained the purpose of the survey beforehand. We conducted a pilot survey with three practitioners to gather feedback on (a) the survey's duration and (b) the clarity and comprehensibility of the assigned tasks. Based on their feedback, we made minor modifications and finalized the survey. We inform participants that the estimated time to complete the survey is 15-20 minutes. The responses from the pilot survey were excluded from the final analysis. Our survey comprises the following five parts.

Consent and Prerequisite. To be eligible for the survey, participants must confirm their consent, agree to data processing, be familiar with the SO, and have experience in programming languages (e.g., Python or Java).

Participants Information. In this section, we collect participants' demographic and professional software development experience.

Evaluation of Inline Comments. Participants are presented with four randomly selected code snippets (Python or Java) with inline comments, one from each quartile generated by standard Gemini. They were asked to evaluate the accuracy, adequacy, conciseness, and usefulness of the inline comments. We provide five options - very bad (1), bad (2), average (3), good (4) and excellent (5).

TABLE III: Experience, profession, and SO usage frequency of participants.

Development Experience (Years)					Professsion			Frequency of SO Usage			
<= 2	3-5	6-10	11-15	>15	SW Developer	Academician	Student	Daily	Weekly	Monthly	Rarely
-	57.1%	42.9%	-	-	64.3%	7.1%	28.6%	64.3%	21.4%	14.3%	-

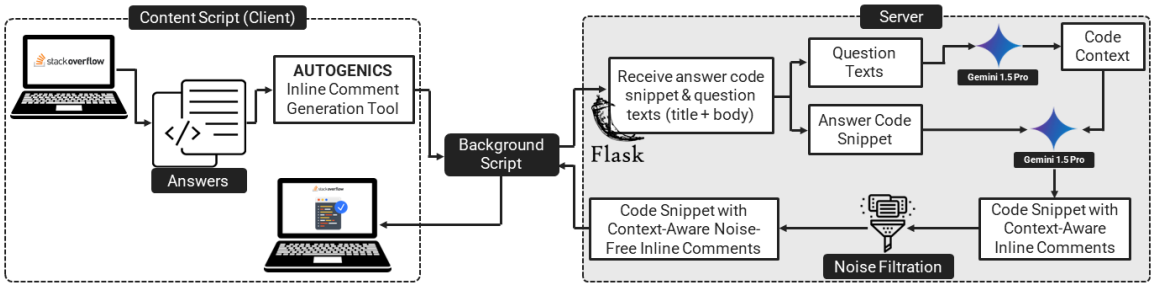


Fig. 4: An overview of AUTOGENICS system architecture.

Tool Support Needs and Preferences. In this section, we inquired about participants' interest in automated tools for generating inline code comments for code snippets in SO answers with five options: not interested at all, not very interested, neutral, somewhat interested, and very interested. We also asked their preference on which types of tools they preferred: web app, browser plugin, IDE extension, or API service.

Participants. We recruit active users of SO as participants (Table III) and select them as follows.

- *Snowball Approach:* We use convenience sampling to bootstrap the snowball [53]. In a collaborative effort, we first contacted a few software developers who were known to us, easily reachable, and working in software companies worldwide. We explained our study goals and shared the online survey with them. We then adopted a snowballing method [54] to disseminate the survey to several of their colleagues with similar experiences.
- *Open Circular:* We circulate the survey to specialized Facebook groups. In particular, we target the groups where professional Python/Java developers discuss their programming problems. We also use LinkedIn to find potential participants because it is one of the largest professional networks.

We recruited 14 participants from countries worldwide (e.g., Canada, Bangladesh) with diverse professions and experience levels and received 14 valid responses (nine for Python + five for Java). Table III summarizes the participants' experience and professions. We then analyze the responses with appropriate tools and techniques based on the question types.

E. AUTOGENICS*: Automated Generation of Inline Code Comments*

During the manual evaluation of inline comments (in RQ1), we identified two issues: (1) LLMs frequently generate comments for unnecessary statements in code snippets (e.g., print and import statements), which developers consider as noise [55, 56], and (2) inline comments often fail to represent the purpose of the statements without additional context. We introduce AUTOGENICS that can address these challenges and generate context-aware, noise-free inline comments for SO code snippets.

Architecture of AUTOGENICS and Context-Aware Comment Generation. AUTOGENICS is an easy-to-use tool that can be integrated with the user's browser, enabling direct interaction with the SO Q&A interface. Figure 4 shows the overview of the system architecture of AUTOGENICS. First, we configure the browser extension by defining its properties and permissions. These include permissions to interact with active tabs on SO and to communicate with the local Flask server [57]. Then, we specify the background service worker and content script to inject functionality into SO pages.

The content script is injected into SO pages to interact with the DOM elements. It locates code snippets within answers and enables a button labeled "AUTOGENICS" next to each code snippet. The script extracts question texts (title + body) to support LLMs with additional context. Upon clicking the "AUTOGENICS" button, the content script sends the candidate code snippet and question texts to the background script. The background script facilitates communication between the content script and the Flask server. It listens for messages from the content script and, upon receiving a request, sends the code snippet and question texts to the Flask server. Subsequently, it waits for the server's response, which includes the generated comments.

This tool's main functionality is achieved by running a local Flask server, which acts as the backend for handling requests from the background script. We configure the server to support Cross-Origin Resource Sharing (CORS) to enable interaction with the SO domain. This setup is crucial because AUTOGENICS needs to send requests from a web page to the local server. Then, we integrate the LangChain framework [58] with Google's `Gemini 1.5 Pro` model. This setup involves loading environment variables, including the Google API key, which is necessary for authenticating requests to the model. First, the tool extracts additional code context from SO question texts (title and body). To achieve this, we employ a specially designed prompt with question texts within LangChain as follows.

Question context extraction prompt (AUTOGENICS): Extract the main context and key points from the following question description of SO: {QUESTION_DESCRIPTION}. This will help understand the purpose and requirements of the code provided in the answers.

Then Gemini processes question texts as input and extracts critical information from them. Later, when combined with code snippets, this contextual information helps better understand the purpose and requirements of the code snippet. This context supports AUTOGENICS to generate context-aware inline comments. To generate context-aware inline comments, we designed the prompt as follows.

Context-Aware Inline comment generation prompt (AUTOGENICS): Generate inline comments for the following code snippet: {CODE_SNIPPET}, considering the provided question context: {CODE_CONTEXT}. An inline comment is a single-line comment typically used to explain or clarify a specific line of code. It starts with // for Java and # for Python. Ensure that you only generate inline comments and do not alter the existing code. Avoid adding any comments at the class or method level; focus only on inline comments.

Once the code snippets with comments are received, the background script sends them back to the content script. Upon receipt, the content script inserts them directly into the SO page immediately after the corresponding code snippet within the answer. The comments are presented in a visually distinctive manner, ensuring they are easily readable and recognizable for users aiming to comprehend the code snippet.

Noise Filtering Mechanism. To eliminate noisy comments, we implement a filtering mechanism. This mechanism utilizes regular expressions to detect and remove inline comments from statements that are often treated as noise [59, 60]. Such statements include basic import statements, function definitions, control flow statements, and other typical code patterns that typically do not require explanations (please refer to Table IV for the list of considered patterns). If any specified patterns are matched, we check whether that statement has an inline comment. If found, we discard that inline comment.

Effectiveness Evaluation of AUTOGENICS-generated comments. We randomly select 40 code snippets (20 Python and 20 Java, five from each quartile) from our selected 400 snippets. We generate inline comments for these snippets using AUTOGENICS and manually evaluate their effectiveness using the same four metrics as in RQ1.

TABLE IV: Frequently observed statement patterns to filter out noisy inline comments for Java and Java.

Statement Group	Patterns
Basic import and print statements	print(), import, from [module] import, System.out.print, System.out.println
Function and class definitions	def, class, public class, private class, protected class
Access modifiers	public, private, protected
Common control flow keywords	return, break, continue
Control structures	if, for, while, else, elif, switch, case, default
Variable declarations	var, let, const, int, float, double, String, boolean

IV. Effectiveness Evaluation of LLMs-Generated Inline Comments (RQ1)

TABLE V: Evaluation summary of Gemini-generated inline comments (**M** = *Mean*, **Med** = *Median*; **Q1-Q4:** *Quartiles 1-4*).

Language	Quartile	Accuracy		Adequacy		Conciseness		Usefulness	
		M	Med	M	Med	M	Med	M	Med
Python	Q1	4.65	5	4.12	4	4.36	4	3.94	4
	Q2	4.70	5	4.16	4	4.26	4	4.40	4
	Q3	4.88	5	4.24	4	4.14	4	4.60	5
	Q4	4.74	5	4.34	4	3.92	4	4.76	5
Java	Q1	4.80	5	4.14	4	4.40	4	3.86	4
	Q2	4.82	5	4.26	4	4.32	4	4.36	4
	Q3	4.86	5	4.32	4	4.24	4	4.56	5
	Q4	4.70	5	4.38	4	3.96	4	4.72	5

TABLE VI: Evaluation summary of our case study with GPT-4 in generating inline comments (**M** = *Mean*, **Med** = *Median*; **Q1-Q4:** *Quartiles 1-4*).

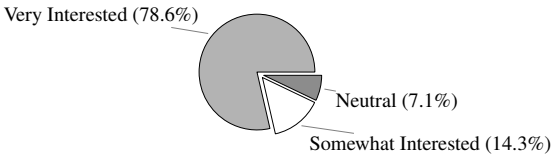
Language	Quartile	Accuracy		Adequacy		Conciseness		Usefulness	
		M	Med	M	Med	M	Med	M	Med
Python	Q1	4.8	5.0	3.8	4.0	4.4	4.0	3.8	4.0
	Q2	4.2	4.0	3.4	3.0	4.0	4.0	3.4	3.0
	Q3	4.4	4.0	3.6	4.0	4.0	4.0	3.6	4.0
	Q4	4.2	4.0	3.4	3.0	3.6	4.0	4.2	4.0
Java	Q1	4.6	5.0	3.6	4.0	4.2	4.0	3.6	4.0
	Q2	4.2	4.0	3.4	3.0	4.0	4.0	3.6	4.0
	Q3	4.4	4.0	3.4	3.0	4.0	4.0	3.8	4.0
	Q4	4.2	4.0	3.4	3.0	3.8	4.0	4.0	4.0

In this section, we manually analyze the effectiveness of LLMs-generated inline code comments using four popular metrics - accuracy adequacy, conciseness, and usefulness. Table V summarizes the results when we generate inline comments using the Gemini model. According to our analysis, the accuracy of the inline comments in Python code snippets is consistently high across all quartiles. Mean accuracy ranges from 4.65 (first quartile) to 4.88 (third quartile), with a steady median of five. Interestingly, we see an upward trend in the accuracy when LOC increases from the first to the third quartile. However, the accuracy score goes slightly downward in the fourth quartile. We find similar results for the inline comments in Java code snippets. Such findings suggest that LLMs (e.g., Gemini) can generate more accurate inline comments for longer code snippets. However, accuracy could decline when the LOC is very high, possibly due to the code's increased complexity and context dependence.

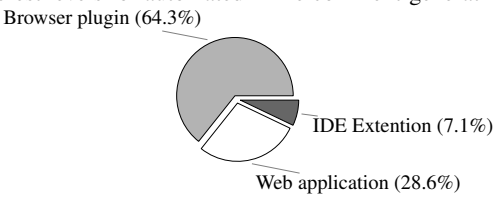
Overall, the adequacy score exceeds four for all quartiles (Python + Java), with a median value of four. We observe a consistent, slight improvement in the adequacy score as the LOC increases. As code length increases, there is more complexity to explain, which might motivate LLMs to provide more detailed inline comments for better understanding. Therefore, we see an opposite trend between conciseness and adequacy. While the median values remain consistent at four, mean scores slightly decrease in the higher quartiles. For example, we observe relatively verbose inline comments within lengthy code snippets. However, we observe a similar trend as adequacy when evaluating the usefulness metric.

TABLE VII: Evaluation summary of inline comments by survey participants (**1** = *Very Bad*, **2** = *Bad*, **3** = *Average*, **4** = *Good*, **5** = *Excellent*; **Q1-Q4**: *Quartiles 1-4*).

Language	Quartile	Accuracy (% of Participants)					Adequacy (% of Participants)					Conciseness (% of Participants)					Usefulness (% of Participants)				
		1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
Python	Q1	-	-	-	66.7	33.3	-	-	66.7	33.3	-	-	-	-	22.2	77.8	-	8.1	69.7	22.2	-
	Q2	-	-	-	33.3	66.7	-	-	33.3	66.7	-	-	-	-	66.7	33.3	-	-	33.3	66.7	-
	Q3	-	-	-	22.2	77.8	-	-	-	44.4	55.6	-	33.3	66.7	-	-	-	-	-	33.3	66.7
	Q4	-	-	-	55.6	44.4	-	-	-	22.2	77.8	22.2	66.7	11.2	-	-	-	-	-	22.2	77.8
Java	Q1	-	-	-	80	20	-	-	80	20	-	-	-	-	20	80	-	10	70	20	-
	Q2	-	-	-	40	60	-	-	40	60	-	-	-	-	80	20	-	-	20	80	-
	Q3	-	-	-	20	80	-	-	-	40	60	-	20	60	20	-	-	-	-	40	60
	Q4	-	-	-	60	40	-	-	-	20	80	40	60	-	-	-	-	-	-	20	80



(a) Interest levels for automated inline comment-generating tool.



(b) Tool preferences.

Fig. 5: Interest levels and preferences for an automated inline comment-generating tool.

We conducted a case study to evaluate the performance of the `GPT-4` model in generating inline comments. We randomly select 40 code snippets from our dataset (20 Python + 20 Java, evenly distributed across quartiles) to generate inline comments utilizing `GPT-4`. We then manually evaluate the four metrics. TableVI summarizes the results of each metric. The accuracy and conciseness of the `Gemini 1.5 Pro` model are comparable to those of `GPT-4`. However, there were notable differences in adequacy and usefulness scores, where `GPT` underperforms relative to `Gemini`. Interestingly, `GPT` generates more accurate inline comments for Python code snippets of shorter lengths (i.e., within the first quartile). Therefore, selecting the `Gemini 1.5 Pro` model for deploying a tool to generate inline comments is cost-effective and offers performance comparable to or exceeding that of `GPT-4`.

Summary RQ1. The `Gemini 1.5 Pro` model generates highly accurate and adequate inline comments, especially for longer code snippets, while `GPT-4` excels with shorter Python snippets. Overall, `Gemini 1.5 Pro` is cost-effective and performs comparably or better than `GPT-4`, making it a reasonable choice for generating inline comments.

V. Practitioners' Perspective on LLM-Generated Comment Effectiveness & Tool Preference (RQ2)

In this section, we survey 14 SO users to perceive the effectiveness of LLM-generated inline comments and tool support preferences for generating them automatically.

A. Effectiveness Evaluation of Inline Comments by Survey Participants.

Table VII shows the effectiveness evaluation summary by the survey participants, which closely aligns with the results of our manual evaluation (Table V). These consistent results enhance the confidence of our evaluation.

The number of participants who rated the comment's accuracy as Excellent increased from the first to the third quartile, followed by a slight decline, similar to what we observed in RQ1. The adequacy of inline comments for the code snippets was primarily rated as Average (3) for the first quartile and Good (4) for the second quartile. However, it was mostly assessed as Excellent for the third and fourth quartiles. For example, 77.8% of the participants rated the adequacy of inline comments for the fourth quartile's code snippets as Excellent. Similar findings are shown for Usefulness.

On the contrary, the inline comments of the code snippets from the upper two quartiles were evaluated as less concise (e.g., Good or below ratings). The opposite results are found in the lower two quartiles. For example, 77.8% of the participants rated the inline comments of the code snippet from the first quartile as Excellent (5).

B. Tool Support Needs and Preferences.

As shown in Fig. 5a, approximately 79% of the participants expressed strong interest in tool support for generating inline comments for SO answer code snippets. No participants selected the options 'Not Very Interested' or 'Not Interested At All'. These results indicate the participants' high demand for an inline comments generation tool.

Fig. 5b shows tool supports preference pie chart. For example, 64.3% of the participants prefer an inline comments generator tool as a browser plugin, while 28.6% prefer a web application. Such preference encourages us to consider developing tool support as a browser plugin for generating inline comments.

Summary RQ2. The majority of participants found the LLM-generated inline comments effective and expressed a strong demand for tool support. In particular, they preferred an automated inline commenting tool as a browser plugin on the SO site.

TABLE VIII: Evaluation Summary of conventional and context-aware noise-free inline comments (**M** = *Mean*, **Med** = *Median*; **WO Context** = *Without Context*, **W Context** = *With Context*; **Q1-Q4**: *Quartiles 1-4).*

Language	Quartile	Accuracy				Adequacy				Conciseness				Usefulness			
		WO Context		W Context		WO Context		W Context		WO Context		W Context		WO Context		W Context	
		M	Med	M	Med	M	Med	M	Med	M	Med	M	Med	M	Med	M	Med
Python	**Q1**	4.4	4.0	4.8	5.0	3.4	3.0	4.2	4.0	4.4	4.0	4.6	5.0	3.8	4.0	4.4	4.0
	Q2	4.4	4.0	4.8	5.0	3.6	4.0	4.4	4.0	4.0	4.0	4.4	4.0	4.2	4.0	4.6	5.0
	Q3	4.8	5.0	5.0	5.0	3.8	4.0	4.4	4.0	4.0	4.0	4.2	4.0	4.2	4.0	4.6	5.0
	Q4	4.6	5.0	4.8	5.0	4.2	4.0	4.6	5.0	3.8	4.0	4.0	4.0	4.4	4.0	4.8	5.0
Java	Q1	4.4	4.0	4.8	5.0	3.8	4.0	4.2	4.0	4.0	4.0	4.6	5.0	3.6	4.0	4.2	4.0
	Q2	4.6	5.0	5.0	5.0	3.8	4.0	4.4	4.0	4.0	4.0	4.4	4.0	4.0	4.0	4.4	4.0
	Q3	4.6	5.0	5.0	5.0	4.0	4.0	4.6	5.0	3.8	4.0	4.4	4.0	4.4	4.0	4.6	5.0
	Q4	4.4	4.0	4.8	5.0	4.2	4.0	4.6	5.0	3.6	4.0	4.2	4.0	4.6	5.0	4.8	5.0

VI. AUTOGENICS: A BROWSER PLUGIN TO GENERATE CONTEXT-AWARE NOISE-FREE INLINE COMMENTS (RQ3)

In this section, we compare the effectiveness between comments generated without context by Gemini and comments generated with context by the AUTOGENICS tool. AUTOGENICS considers additional code context from question texts and filters out noise. Table VIII summarizes the results between standard Gemini and AUTOGENICS.

For Python, the evaluation demonstrates significant enhancements when additional code context is incorporated. For example, scores of 'Accuracy' without context are consistent across quartiles, ranging from 4.4 to 4.8. In contrast, context-aware scores increase from 4.8 to 5.0, with medians consistently at 5.0. 'Adequacy' scores without context are moderate, ranging from 3.4 to 4.2, but they improve significantly with context and achieve mean scores ranging from 4.2 to 4.6, with a median of 4.0. These findings highlight the crucial role of contextual information from SO questions in generating relevant comments. For 'Conciseness', Python inline comments without context score around 3.8 to 4.4, indicating they are fairly concise. Context-aware comments maintain or slightly improve conciseness, scoring between 4.0 and 4.6 with almost similar median values. The usefulness of inline comments shows a consistent pattern—scores range from 3.8 to 4.4 without context. However, scores improve significantly (ranging from 4.4 to 4.8) with context, often achieving median scores of 5.0. These results clearly show the advantages of additional code context in generating inline comments to enhance their effectiveness. Based on our analysis, we observe a similar enhancement in the effectiveness of inline comments for Java code snippets when incorporating contextual information.

Additional context enhances AUTOGENICS's ability to align inline comments with the intended functionality and logic (see Table VIII). Such context improves accuracy by ensuring relevant comments and correctly describing code operations. Understanding context also allows AUTOGENICS to include critical insights that might otherwise be overlooked, ensuring thorough explanations of code complexities. AUTOGENICS also filters out unnecessary distractions. Consider the examples in Fig. 6, which demonstrate how incorporating context enhances the ability of AUTOGENICS to generate more effective inline comments. First, AUTOGENICS extract the question texts. It then prompts Gemini with the question texts to produce a structured context. For example, it produces context, key points, potential causes of the error, and important points to resolve errors for the given question (see Fig. 6). Next, AUTOGENICS passes the candidate answer code snippets and this context to generate inline comments. In this scenario, AUTOGENICS identifies the inherent meaning of `Ytrain1` as a 'training set'. It gets this contextual information from the context, **"However, when applied to the training subset 'Ytrain' after splitting the data into training and testing sets, an error occurs on the line 'T1[i, Ytrain[i]] = 1'."**. Besides, AUTOGENICS properly pinpoints `T1` as an indicator matrix and `K` as the number of classes. It fetches the context for the `T1` from this line- **"The code snippet aims to convert a target variable 'Y' into an indicator matrix."**, and `K` variable from this line- **"'K' represents the number of classes (9), and the code uses a for loop to iterate through each sample in 'Ytrain'."**. In addition, the filtration mechanism of AUTOGENICS filters out comments for the `for` and `print` statements, where the standard Gemini model generates comments.

Summary RQ3. AUTOGENICS is a user-friendly tool designed to be integrated with the SO Q&A site. It can generate context-aware, noise-free inline comments that significantly improve the overall quality of standard LLM-generated comments.

VII. DISCUSSION

In this section, we discuss the key findings and implications of this study.

A. Key Findings

Consistently High Accuracy Showcases LLM Mastery. The LLM's consistently high accuracy ratings for Python and Java demonstrate its robust understanding of programming concepts across different languages and code lengths correctly. LLMs achieve peak comment accuracy in the third quartile of code length, striking the ideal balance between context richness and manageable complexity. Additionally, its performance benefits from incorporating additional code context.

Dilemma of Clarity Vs. Conciseness. As code complexity increases, the LLM faces challenges keeping comments concise and often provides more detailed explanations to describe code snippets accurately. Such a scenario reflects the model's

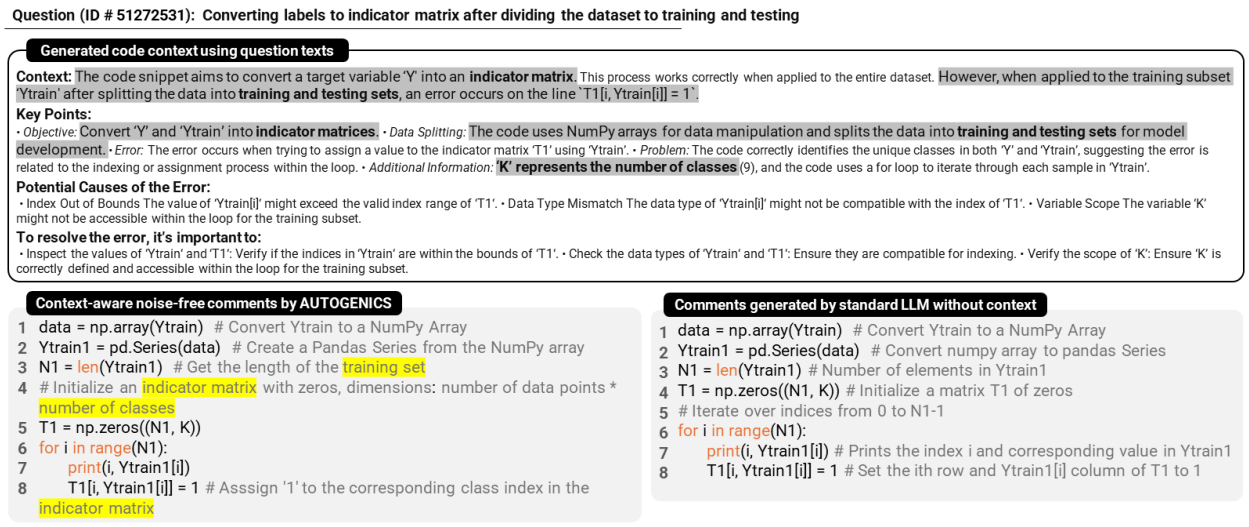


Fig. 6: Example of how AUTOGENICS uses context to generate effective inline comments.

effort to balance clarity and thoroughness, emphasizing the importance of capturing the essence of the code without oversimplification through more extensive comments.

Insights into LLM's Context-Driven Commenting Process. LLMs prioritize understanding entire code contexts before generating inline comments. Thus, they can generate high-quality comments. As shown in Listing 1, the Gemini generates an inline comment indicating that 'a' is a PyTorch tensor. Such comments involve analyzing common coding patterns and leveraging extensive training data to accurately infer and explain code elements.

```
a_n = a.numpy()  # Convert the PyTorch tensor to a NumPy array
# Apply a function along the 2nd axis, summing the powers of 2 of the non-zero elements in each row
a_n = np.apply_along_axis(func1d=lambda x: np.sum(np.power(2,np.where(x==1))[0]), axis=2, arr=a_n)
a = torch.Tensor(a_n)  # Convert the NumPy array back to a PyTorch tensor
```

Listing 1: Python code example illustrating the LLM's context-driven inline commenting process.

B. Implications

Our research findings on automated inline comment generation for SO answer code snippets and the development of AUTOGENICS will provide a basis for future studies. Future **researchers** can extend our findings to enhance automated code comprehension and documentation capabilities for online code snippets. Additionally, our approach emphasizes the potential of integrating automated tools into coding environments to improve developer productivity and coding efficiency.

AUTOGENICS, a browser plugin integrated with SO, demonstrates the feasibility and benefits of real-time comment assistance for **developers** (e.g., SO users) to generate context-aware inline comments in code snippets. It will promote faster comprehension of unfamiliar code, particularly aiding novices in understanding and integrating solutions more accurately.

AUTOGENICS benefits **educators**, including programming bloggers and tutorial makers, by (a) providing instant feedback on code clarity, (b) ensuring their code examples are well-documented and easy to understand, and thus (c) enhancing educational content. **Stack Overflow site owners** can integrate AUTOGENICS to improve user experience by enabling real-time inline comments on code snippets, promoting better code readability and documentation.

VIII. Threats to Validity

External Validity is related to how broadly our results can be generalized. Our results may not be generalized to all SO answers code snippets. To mitigate this threat, we analyze statistically significant samples from two popular programming languages - Java and Python. Java is statically typed, whereas Python is a dynamically typed programming language. We also categorize code snippets based on LOC and take samples evenly distributed across quartiles. We see that the results from both languages are consistent. Thus, we believe that our insights can be generalized to other programming languages. Moreover, we investigate a wide variety of answers to different types of programming problems in order to combat potential bias in our results. However, we caution readers to refrain from over-generalizing our results. Another threat to generalizability is the use of specific language models. To address this, we conducted a case study with GPT-4 in addition to Gemini 1.5 Pro, thereby mitigating the threat.

Threats to *internal validity* relate to experimental errors and biases [61]. We manually evaluate the effectiveness of inline comments using four metrics that could introduce bias. However, the agreement between the two annotators was almost perfect (i.e., $\kappa = 0.94$), which ensures the robustness and consistency of our evaluations. We surveyed 14 developers who evaluated the effectiveness of the inline comments. Their results were consistent with ours, further validating our evaluation.

Threats to *construct validity* relate to the suitability of evaluation metrics. To mitigate this threat, we evaluate the effectiveness of the inline comments using four appropriate metrics - accuracy, adequacy, conciseness, and usefulness. *Statistical Conclusion* threats concern the fact that the data is sufficient to support the claims. We considered statistically significant samples in our result analyses.

Snowball sampling relies on referrals and may have a sampling bias. However, we also selected participants using an open circular approach and collected their responses anonymously. Table III shows that our participants have diverse experiences and professions. Such diversity offers validity and applicability to our survey findings.

IX. Related Work

Several studies investigate block-level comments, which summarize source code and provide a high-level overview of the purpose and logic of a block of code [1, 2, 18, 62, 63]. Sridhara et al. [62] introduced a technique to identify sequences of statements, conditions, and loops in code that could be summarized into higher-level actions. Then, they generated descriptions for these segments using their templates. Wong et al. [1] developed a method to extract code descriptions from a programming Q&A site (e.g., SO). Then, they utilized these insights to produce comments for equivalent code segments in open-source projects. Researchers also utilized code clone detection techniques to find and reuse comments from code libraries in open-source software [2].

The block comment generation domain has also benefited from learning-based approaches, which treat code as text sequences or interpret Abstract Syntax Trees (AST) as sequences. For example, Iyer et al. [18] introduced CODE-NN, an LSTM-based neural network model that takes code sequences as input and produces sequences of comment tokens. On the other hand, Huang et al. [63] combined heuristic rules with learning-based techniques to develop a reinforcement learning strategy for generating block comments.

Numerous studies investigate method-level comments, which describe a method's overall intent, parameters, return values, and functionality. Sridhara et al. [64] applied the Software Word Usage Model (SWUM) and heuristic-based techniques to select keywords from code. They create templates to explain Java methods. Vassallo et al. [65] introduced a technique to extract method comments leveraging Q&A discussion of SO.

Several studies highlight the effectiveness of Abstract Syntax Trees (ASTs) in capturing structural properties in order to improve the quality of method-level comments [19, 20, 21, 22, 23]. These techniques largely depend on programming language and language-specific dictionaries. To address these issues, Moore et al. [24] designed a CNN model treating code as character sequences to manage dictionary size effectively. Li et al. [66] introduced SeCNN, which integrates lexical and syntactic details to improve comment quality and handle longer dependencies in code.

Li et al. [67] introduced SeTransformer, a transformer-based architecture that enhances upon CNNs and RNNs with a self-attention mechanism for simultaneous text and structural feature analysis of code. On the other hand, Yang et al. [68] developed ComFormer, integrating Transformer models with a fusion method to improve comment quality. Meanwhile, Xu et al. [69] explored local and global encoders with Graph Attention Networks for contextual information in comment generation. Kuang et al. [70] proposed GTrans, combining Graph Neural Networks with Transformers for comprehensive code representation.

The studies mentioned above focus on generating block-level and method-level comments using techniques such as Information Retrieval, templating, or Deep Learning approaches. To our knowledge, our study pioneers the use of LLMs for generating inline comments on code snippets found in SO answers. We demonstrate the effectiveness of these comments through manual analysis and user study for two widely used programming languages. We introduce AUTOGENICS, a tool for generating context-aware, noise-free inline comments. It has the potential to significantly enhance code comprehension and assist millions of developers in effectively utilizing and reusing code resources.

X. Conclusion

Inline code comments play a crucial role in enhancing code comprehension, readability, and reusability, particularly in programming Q&A sites like SO. First, we investigate the capability of standard LLMs (e.g., Gemini) to generate effective inline code comments. We randomly select 400 code snippets (200 Python + 200 Java) extracted from accepted answers on SO and employ standard LLMs to generate inline comments. We manually assess four key metrics—accuracy, adequacy, conciseness, and usefulness of the comments. Additionally, we surveyed 14 software developers who are active users of SO to perceive the effectiveness of these inline comments. Our evaluation demonstrates the promise of LLMs in generating inline comments. However, it has a few limitations, such as a lack of effectiveness for shorter code snippets and the presence of noisy comments. We then introduced AUTOGENICS, a tool leveraging LLM as a browser plugin to automatically generate context-aware, noise-free inline comments for code snippets in SO answers. It can overcome the limitations of standard LLMs by utilizing additional code context from question texts and a noise filtration mechanism. By optimizing comments and removing irrelevant noise, AUTOGENICS aims to improve the overall usability of code snippets on SO, facilitating better learning and reuse practices among developers.

In future studies, we intend to explore the effectiveness of AUTOGENICS across different programming languages and Q&A platforms. Additionally, we plan to conduct an expert survey to gather user feedback on AUTOGENICS, aiming to enhance its usability based on their insights.

Acknowledgment. This research is supported in part by the industry-stream NSERC CREATE in Software Analytics Research (SOAR).

REFERENCES

[1] Edmund Wong, Jinqiu Yang, and Lin Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 562–567. IEEE, 2013.

[2] Edmund Wong, Taiyue Liu, and Lin Tan. Clocom: Mining existing source code for automatic comment generation. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 380–389. IEEE, 2015.

[3] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. Retrieval-based neural source code summarization. In *ACM/IEEE 42nd International Conference on Software Engineering*, pages 1385–1397, 2020.

[4] Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Would static analysis tools help developers with code reviews? In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 161–170. IEEE, 2015.

[5] Bradley L Vinz and Letha H Etzkorn. Improving program comprehension by combining code understanding with comment understanding. *Knowledge-Based Systems*, 21(8):813–825, 2008.

[6] M-A Storey, L-T Cheng, Janice Singer, M Muller, Del Myers, and Jody Ryall. How programmers can turn comments into waypoints for code navigation. In *2007 IEEE International Conference on Software Maintenance*, pages 265–274. IEEE, 2007.

[7] István Kádár, Péter Hegedus, Rudolf Ferenc, and Tibor Gyimóthy. A code refactoring dataset and its assessment regarding software maintainability. In *2016 IEEE 23rd International conference on software analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 599–603. IEEE, 2016.

[8] Gang Huang, Hong Mei, and Fu-Qing Yang. Runtime recovery and manipulation of software architecture of component-based systems. *Automated Software Engineering*, 13:257–281, 2006.

[9] Gang Huang, Yun Ma, Xuanzhe Liu, Yuchong Luo, Xuan Lu, and M Brian Blake. Model-based automated navigation and composition of complex service mashups. *IEEE Transactions on Services Computing*, 8(3):494–506, 2014.

[10] Yuan Huang, Nan Jia, Junhuai Shu, Xinyu Hu, Xiangping Chen, and Qiang Zhou. Does your code need comment? *Software: Practice and Experience*, 50(3):227–245, 2020.

[11] Yuan Huang, Xinyu Hu, Nan Jia, Xiangping Chen, Yingfei Xiong, and Zibin Zheng. Learning code context information to predict comment locations. *IEEE Transactions on Reliability*, 69(1):88–105, 2019.

[12] Qingying Chen and Minghui Zhou. A neural framework for retrieval and summarization of source code. In *33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 826–831, 2018.

[13] Yuding Liang and Kenny Zhu. Automatic generation of text descriptive comments for code blocks. In *AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[14] Saikat Mondal, Mohammad Masudur Rahman, and Chanchal K Roy. Do subjectivity and objectivity always agree*f* a case study with stack overflow questions. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 389–401. IEEE, 2023.

[15] Denae Ford and Chris Parnin. Exploring causes of frustration for software developers. In *2015 IEEE/ACM 8th international workshop on cooperative and human aspects of software engineering*, pages 115–116. IEEE, 2015.

[16] Xing Hu, Xin Xia, David Lo, Zhiyuan Wan, Qiuyuan Chen, and Thomas Zimmermann. Practitioners' expectations on automated code comment generation. In *44th International Conference on Software Engineering*, pages 1693–1705, 2022.

[17] Stack Overflow. StackExchage API, Accessed on: May 2024. URL http://data.stackexchange.com/stackoverflow.

[18] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *54th Annual Meeting of the Association for Computational Linguistics 2016*, pages 2073–2083. Association for Computational Linguistics, 2016.

[19] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *26th conference on program comprehension*, pages 200–210, 2018.

[20] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

[21] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering*, 25:2179–2217, 2020.

[22] Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. Automatic source code summarization with extended tree-lstm. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2019.

[23] Alexander LeClair, Siyuan Jiang, and Collin McMillan. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 795–806. IEEE, 2019.

[24] Jessica Moore, Ben Gelman, and David Slater. A convolutional neural network for language-agnostic source code summarization. *arXiv preprint arXiv:1904.00805*, 2019.

[25] Wenhua Wang, Yuqun Zhang, Yulei Sui, Yao Wan, Zhou Zhao, Jian Wu, S Yu Philip, and Guandong Xu. Reinforcement-learning-guided source code summarization using hierarchical attention. *IEEE Transactions on software Engineering*, 48(1): 102–119, 2020.

[26] Luca Pascarella, Magiel Bruntink, and Alberto Bacchelli. Classifying code comments in java software systems. *Empirical Software Engineering*, 24(3):1499–1537, 2019.

[27] Sa Gao, Chunyang Chen, Zhenchang Xing, Yukun Ma, Wen Song, and Shang-Wei Lin. A neural model for method name generation from functional description. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 414–421. IEEE, 2019.

[28] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. Summarizing source code with transferred api knowledge. 2018.

[29] Saikat Mondal, Mohammad Masudur Rahman, Chanchal K Roy, and Kevin Schneider. The reproducibility of programming-related issues in stack overflow questions. *Empirical Software Engineering*, 27(3):62, 2022.

[30] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. 2024.

[31] Junjie Zhao, Xiang Chen, Guang Yang, and Yiheng Shen. Automatic smart contract comment generation via large language models and in-context learning. *Information and Software Technology*, page 107405, 2024.

[32] Jiho Shin, Clark Tang, Tahmineh Mohati, Maleknaz Nayebi, Song Wang, and Hadi Hemmati. Prompt engineering or fine tuning: An empirical assessment of large language models in automated software engineering tasks. *arXiv preprint arXiv:2310.10508*, 2023.

[33] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. Coditt5: Pretraining for source code and natural language editing. In *37th IEEE/ACM International*

Conference on Automated Software Engineering, pages 1–12, 2022.

[34] OpenAI. Gpt-4, 2024. URL https://openai.com/index/gpt-4//.

[35] Anonymous. Replication package, 2024. URL https://github.com/replication-pckg/AUTOGENICS.

[36] JonathanM. Spark handle json with dynamically named subschema, 2023. URL https://stackoverflow.com/questions/76313809.

[37] Sarah Boslaugh. *Statistics in a nutshell: A desktop quick reference*. "O'Reilly Media, Inc.", 2012.

[38] Saikat Mondal, Mohammad Masudur Rahman, and Chanchal K Roy. Can we identify stack overflow questions requiring code snippets? investigating the cause & effect of missing code snippets. 2024.

[39] Google DeepMind. Gemini 1.5 pro, 2024. URL https://deepmind.google/technologies/gemini/.

[40] Anand Anand Das. Gemini 1.5 pro vs gpt-4 turbo benchmarks, 2024. URL https://bito.ai/blog/gemini-1-5-pro-vs-gpt-4-turbo-benchmarks/.

[41] Xing Hu, Qiuyuan Chen, Haoye Wang, Xin Xia, D. Lo, and Thomas Zimmermann. Correlating automated and human evaluation of code documentation generation quality. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31:1 – 28, 2022. doi: 10.1145/3502853.

[42] Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. Out of the bleu: how should we assess quality of the code generation models? *Journal of Systems and Software*, 203:111741, 2023.

[43] T. V. Dam, M. Izadi, and A. Deursen. Enriching source code with contextual data for code completion models: An empirical study. *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 170–182, 2023. doi: 10.1109/MSR59073.2023.00035.

[44] S. Kovalchuk, Vadim Lomshakov, and Artem Aliev. Human perceiving behavior modeling in evaluation of code generation models. *2nd Workshop on Natural Language Generation, Evaluation, and Metrics (GEM)*, 2022. doi: 10.18653/v1/2022.gem-1.24.

[45] Sepehr Hashtroudi, Jiho Shin, H. Hemmati, and Song Wang. Automated test case generation using code models and domain adaptation. *ArXiv*, abs/2308.08033, 2023. doi: 10.48550/arXiv.2308.08033.

[46] Andrew T Jebb, Vincent Ng, and Louis Tay. A review of key likert scale development advances: 1995–2019. *Frontiers in psychology*, 12:637547, 2021.

[47] Tomoko Nemoto and David Beglar. Likert-scale questionnaires. In *JALT 2013 conference proceedings*, pages 1–8, 2014.

[48] Jacob Cohen. Weighted kappa: nominal scale agreement provision for scaled disagreement or partial credit. *Psychological bulletin*, 70(4):213, 1968.

[49] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[50] Barbara A Kitchenham and Shari L Pfleeger. Personal opinion surveys. In *Guide to advanced empirical software engineering*, pages 63–92. Springer, 2008.

[51] Robert M Groves, Floyd J Fowler Jr, Mick P Couper, James M Lepkowski, Eleanor Singer, and Roger Tourangeau. *Survey methodology*, volume 561. John Wiley & Sons, 2009.

[52] Janice Singer and Norman G. Vinson. Ethical issues in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 28(12):1171–1180, 2002.

[53] Samuel J Stratton. Population research: convenience sampling strategies. *Prehospital and disaster Medicine*, 36(4):373–374, 2021.

[54] Tingting Bi, Xin Xia, David Lo, John Grundy, Thomas Zimmermann, and Denae Ford. Accessibility in software practice: A practitioner's perspective. *arXiv preprint arXiv:2103.08778*, 2021.

[55] Zixuan Song, Xiuwei Shang, Mengxuan Li, Rong Chen, Hui Li, and Shikai Guo. Do not have enough data? an easy data augmentation for code summarization. *2022 IEEE 13th International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*, pages 1–6, 2022. doi: 10.1109/PAAP56126.2022.10010698.

[56] Anh T. V. Dau, Jin L. C. Guo, and Nghi D. Q. Bui. Docchecker: Bootstrapping code large language model for detecting and resolving code-comment inconsistencies, 2024.

[57] Armin Ronacher. Flask wen framework.

[58] Harrison Chase. Langchain, 2024. URL https://www.langchain.com/.

[59] Jaya Zhané. Python commenting worst practices., Accessed on: June 2024. URL https://realpython.com/python-comments-guide/.

[60] Vivek Singh. Create python comments the right way., Accessed on: June 2024. URL https://kinsta.com/blog/python-comments/.

[61] Y. Tian, D. Lo, and J. Lawall. Automated construction of a software-specific word similarity database. In *Proc. CSMR-WCRE*, pages 44–53, 2014.

[62] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *33rd International Conference on Software Engineering*, pages 101–110, 2011.

[63] Yuan Huang, Shaohao Huang, Huanchao Chen, Xiangping Chen, Zibin Zheng, Xiapu Luo, Nan Jia, Xinyu Hu, and Xiaocong Zhou. Towards automatically generating block comments for code snippets. *Information and Software Technology*, 127:106373, 2020.

[64] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. Towards automatically generating summary comments for java methods. In *25th IEEE/ACM international conference on Automated software engineering*, pages 43–52, 2010.

[65] Carmine Vassallo, Sebastiano Panichella, Massimiliano Di Penta, and Gerardo Canfora. Codes: Mining source code descriptions from developers discussions. In *22nd International Conference on Program Comprehension*, pages 106–109, 2014.

[66] Zheng Li, Yonghao Wu, Bin Peng, Xiang Chen, Zeyu Sun, Yong Liu, and Deli Yu. Secnn: A semantic cnn parser for code comment generation. *Journal of Systems and Software*, 181:111036, 2021.

[67] Zheng Li, Yonghao Wu, Bin Peng, Xiang Chen, Zeyu Sun, Yong Liu, and Doyle Paul. Setransformer: A transformer-based code semantic parser for code comment generation. *IEEE Transactions on Reliability*, 72(1):258–273, 2022.

[68] Guang Yang, Xiang Chen, Jinxin Cao, Shuyuan Xu, Zhanqi Cui, Chi Yu, and Ke Liu. Comformer: Code comment generation via transformer and fusion method-based hybrid code representation. In *2021 8th International Conference on Dependable Systems and Their Applications (DSA)*, pages 30–41. IEEE, 2021.

[69] X Xu, Quzhe Huang, Zheng Wang, Yansong Feng, and Dongyan Zhao. Towards context-aware code comment generation. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 3938–3947. Association for Computational Linguistics, 2020.

[70] Li Kuang, Cong Zhou, and Xiaoxian Yang. Code comment generation based on graph neural network enhanced transformer model for code understanding in open-source software ecosystems. *Automated Software Engineering*, 29(2):43, 2022.